Programming Parallel Computers

Chapter 3: Multithreading with OpenMP

Introduction

We have already used OpenMP parallel for loops **in Chapter 2**, but let us now explore OpenMP in more detail.

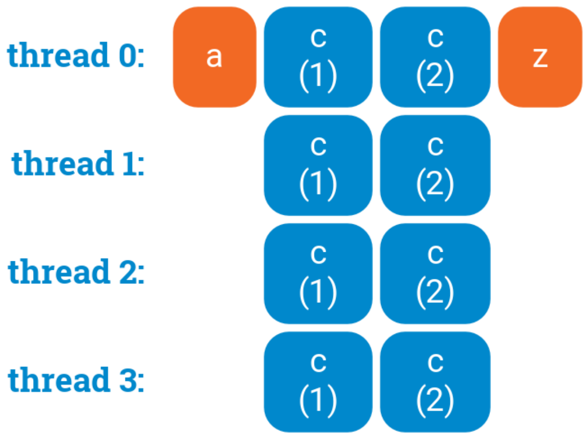
Recall that OpenMP is readily available in the GCC compiler; to use it, we just need to add the right `#pragma omp` directives and compile and link with the command line switch `-fopenmp`.

Basic multithreading construction: parallel regions

Any useful OpenMP program has to use at least one `parallel` region. This is a construction that tells OpenMP that we would like to create multiple threads:

```
a();
#pragma omp parallel
{
    c(1);
    c(2);
}
z();
```

If we execute the above program, the timeline of the execution might look e.g. like this (in the figure, time goes from left to right):



The number of threads is automatically chosen by OpenMP; it is typically equal to the **number of hardware threads that the CPU supports**, which in the case of a low-end CPU is usually the number of CPU cores. In the examples of this chapter we use a 4-core CPU with 4 threads.

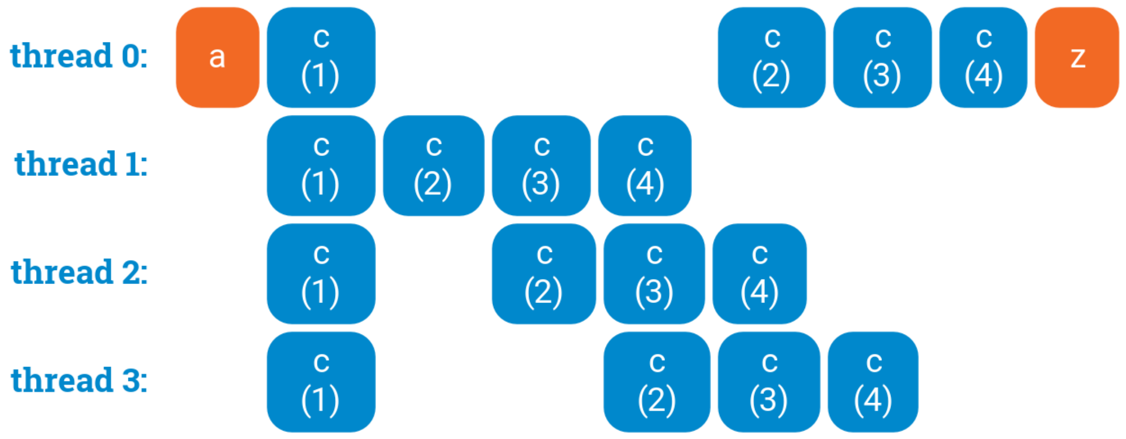
Everything that is contained inside a `parallel` region is executed by all threads. Note that there is no synchronization inside the region unless you explicitly ask for it. Here, for example, different threads might execute at slightly different speeds, and some might start with `c(2)` already while some other threads are still processing `c(1)`. However, OpenMP will automatically wait for all threads to finish their work before continuing the part that comes after the loop — so in the above example we know that when `z()` starts, all threads have finished their work.

Naturally, this example is completely useless by itself as **all threads do the same work**! We will very soon see how to do **worksharing**, i.e., how to tell different threads to do different things. Before getting there, let us first have a look at the main coordination primitive: critical sections.

Critical sections

Whenever we modify a shared resource, we must take care of proper synchronization between the threads. The simplest synchronization primitive is a critical section. A critical section ensures that **at most one thread is executing code that is inside the critical section**. For example, here only one thread is running `c(2)` at any point of time:

```
a();
#pragma omp parallel
{
    c(1);
    #pragma omp critical
    {
        c(2);
    }
    c(3);
    c(4);
}
z();
```



Note that other threads are free to execute e.g. `c(3)` while another thread is running `c(2)`. However, no other thread can start executing `c(2)` while one thread is running it.

As we will discuss **later** in more detail, critical sections also ensure that modifications to shared data in one thread will be visible to the other threads, as long as all references to the shared data are kept inside critical sections. Each thread synchronizes its local caches with global memory whenever it enters or exits a critical section.

Shared vs. private data

Any variable that is declared outside a `parallel` region is **shared**: there is only one copy of the variable, and all threads refer to the same variable. Care is needed whenever you refer to such a variable.

Any variable that is declared inside a `parallel` region is **private**: each thread has its own copy of the variable. Such variables are always safe to use.

If a shared variable is read-only, you can safely refer to it from multiple threads inside the `parallel` region. However, if **any thread ever writes to a shared variable**, then proper coordination is needed to ensure that no other thread is simultaneously reading or writing to it.

Here is an example of the use of shared and private variables:

```
static void critical_example(int v) {

    // Shared variables
    int a = 0;
    int b = v;

    #pragma omp parallel
    {
        // Private variable - one for each thread
        int c;

        // Reading from "b" is safe: it is read-only
        // Writing to "c" is safe: it is private
        c = b;

        // Reading from "c" is safe: it is private
        // Writing to "c" is safe: it is private
        c = c * 10;

        #pragma omp critical
        {
            // Any modifications of "a" are safe:
            // we are inside a critical section
            a = a + c;
        }
    }

    // Reading from "a" is safe:
    // we are outside parallel region
    std::cout << a << std::endl;
}
```

If you run `critical_example(1)` on a machine with 4 threads, it will output “40”. Each thread will set its own private variable `c` to 1, then it will multiply it by 10, and finally each thread will increment the shared variable `a` by 10. Note that all references to `a` — both reading and writing — are kept inside a critical section.

Other shared resources

Note that e.g. I/O streams are shared resources. Any piece of code that **prints something to stdout** has to be kept inside a critical section or outside a `parallel` region.

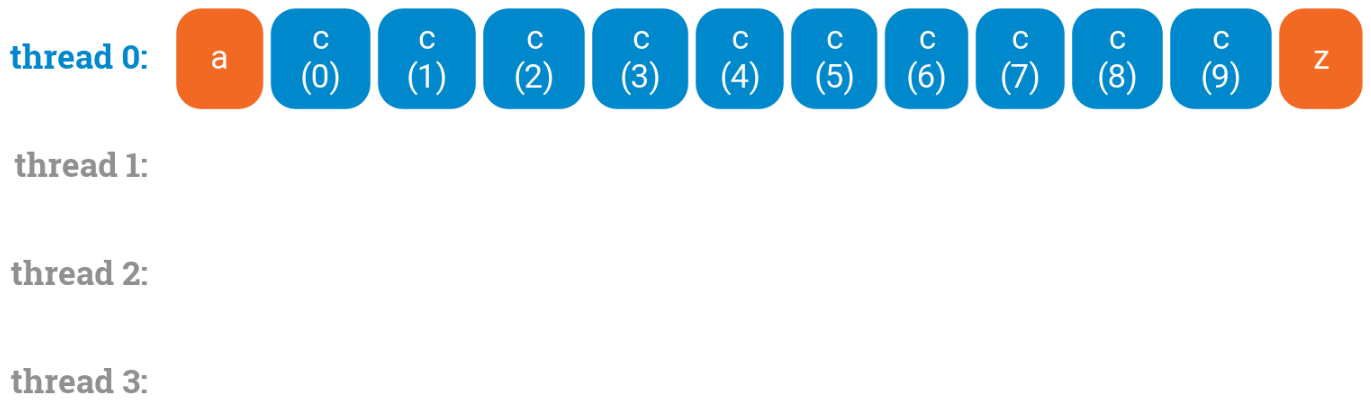
Programming Parallel Computers

Chapter 3: Multithreading with OpenMP

OpenMP parallel for loops

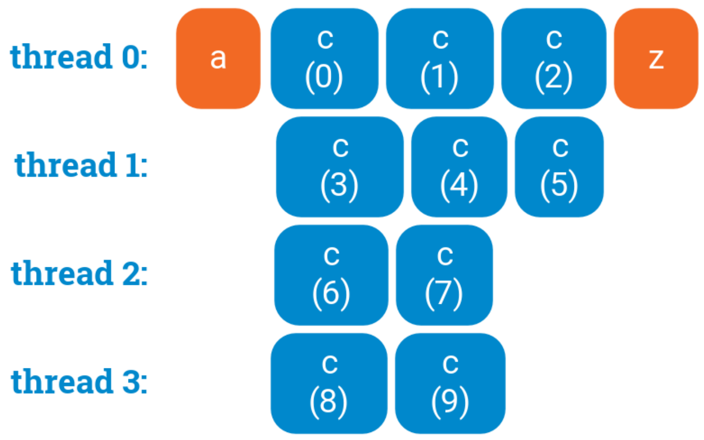
Let us start with a basic example: we would like to do some preprocessing operation `a()`, then we would like to some independent calculations `c(0)`, `c(1)`, …, and finally some postprocessing `z()` once all calculations have finished. In this example, each of the calculations takes roughly the same amount of time. A straightforward for-loop uses only one thread and hence only one core on a multi-core computer:

```
a();
for (int i = 0; i < 10; ++i) {
    c(i);
}
z();
```



With OpenMP parallel for loops, we can easily parallelize it so that we are making a much better use of the computer. Note that OpenMP automatically waits for all threads to finish their calculations before continuing with the part that comes after the parallel for loop:

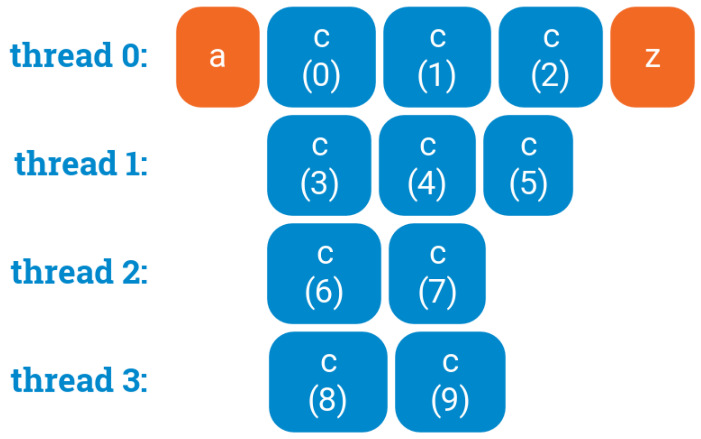
```
a();
#pragma omp parallel for
for (int i = 0; i < 10; ++i) {
    c(i);
}
z();
```



It is just a shorthand

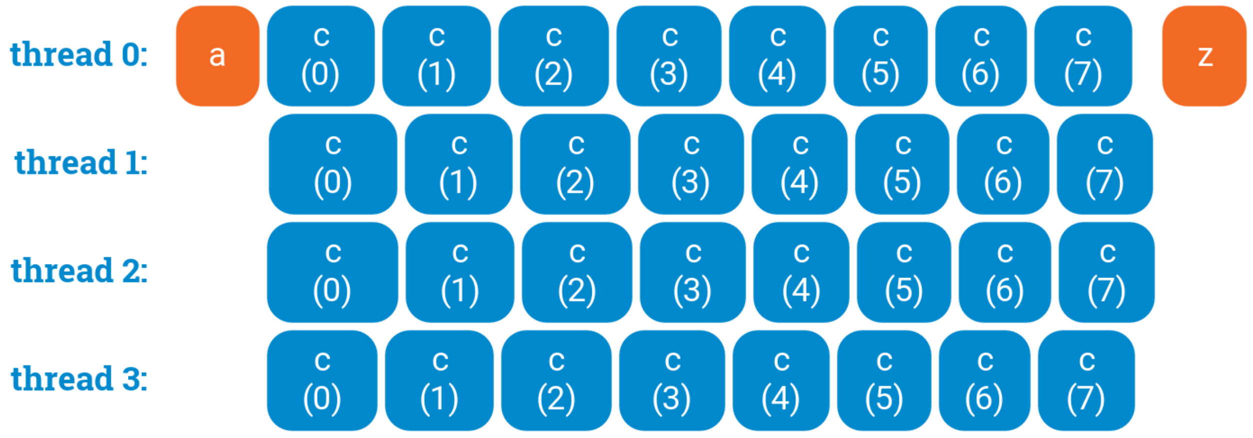
The `omp parallel for` directive is just a commonly-used shorthand for the combination of two directives: `omp parallel`, which declares that we would like to have multiple threads in this region, and `omp for`, which asks OpenMP to assign different iterations to different threads:

```
a();
#pragma omp parallel
{
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
}
z();
```



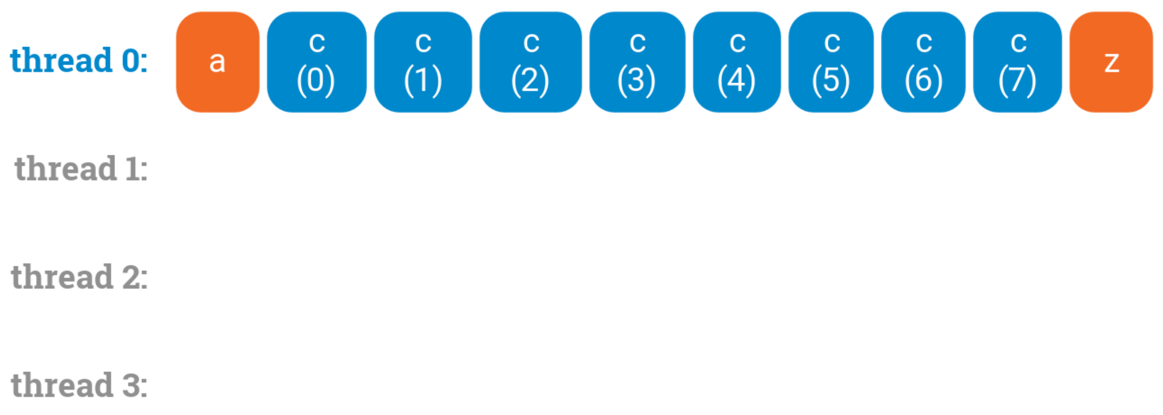
A common mistake is to just use `omp parallel` together with a for loop. This creates multiple threads for you, but it does not do any worksharing — all threads simply run all iterations of the loop, which is most likely not what you want:

```
a();
#pragma omp parallel
for (int i = 0; i < 8; ++i) {
    c(i);
}
z();
```



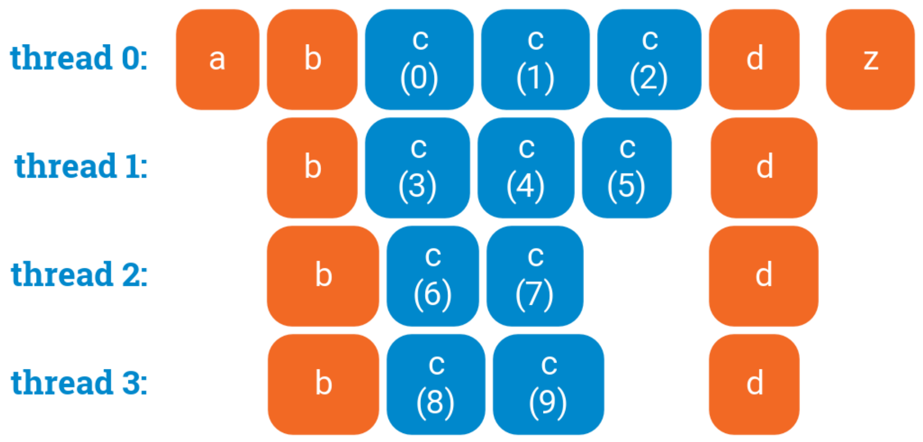
Another common mistake is to use `omp for` alone outside a parallel region. It does not do anything if we do not have multiple threads available:

```
a();
#pragma omp for
for (int i = 0; i < 8; ++i) {
    c(i);
}
z();
```



While in many cases it is convenient to combine `omp parallel` and `omp for` in one directive, please remember that you can always split them. This way it is possible to do some thread-specific preprocessing and postprocessing if needed:

```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



For example, this way you can declare local variables and initialize local data structures in a convenient manner.

Intro | for | nowait | schedule | nested | Hyper-threading | Memory | More | Examples

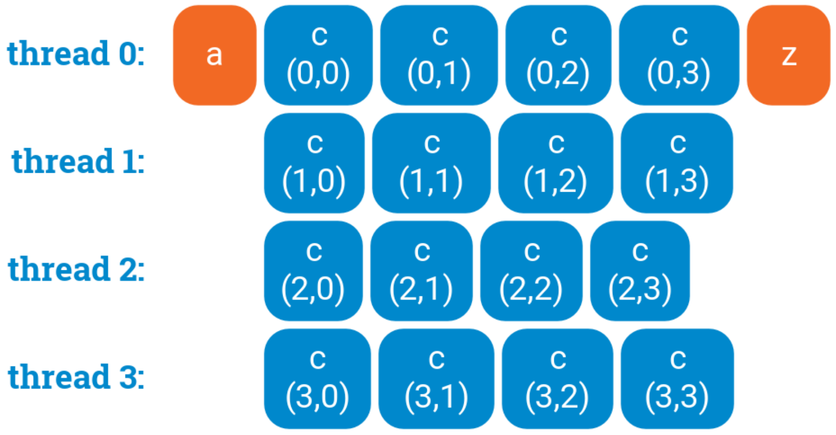
Programming Parallel Computers

Chapter 3: Multithreading with OpenMP

Parallelizing nested loops

If we have nested for loops, it is often enough to simply **parallelize the outermost loop**:

```
a();
#pragma omp parallel for
for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 4; ++j) {
        c(i, j);
    }
}
z();
```

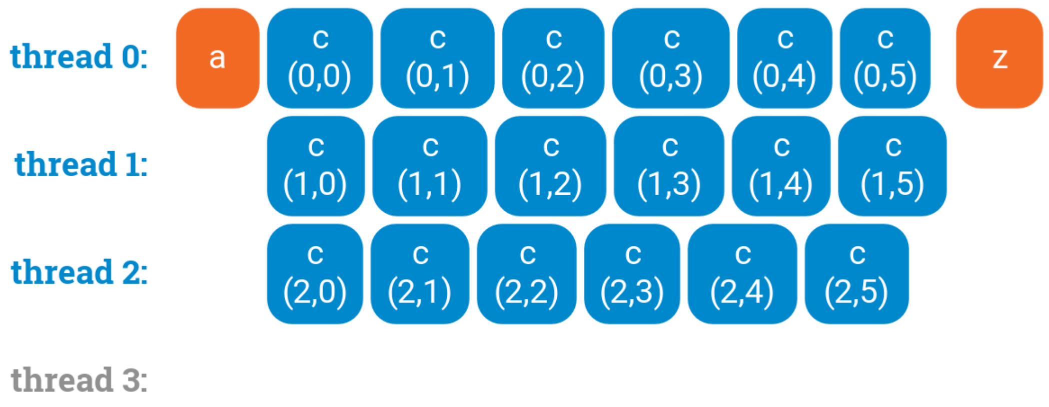


This is all that we need most of the time. You can safely stop reading this part now; in what follows we will just discuss what to do in some rare corner cases.

Challenges

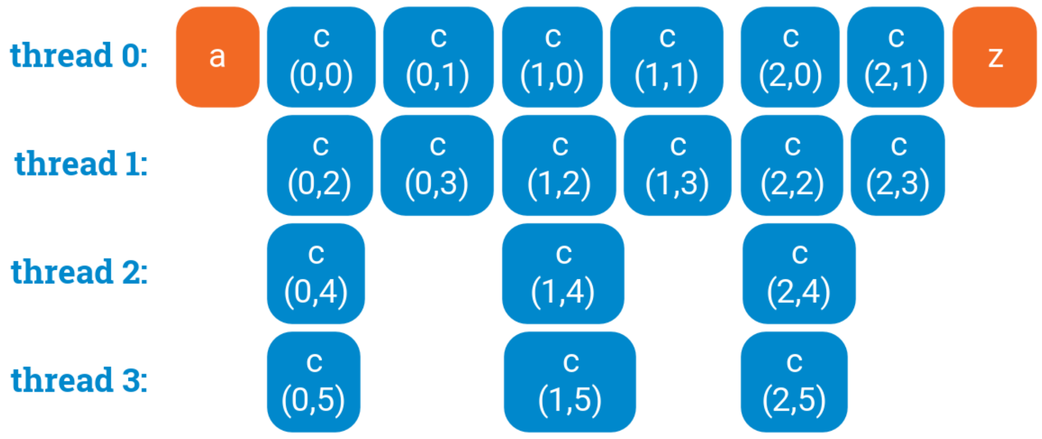
Sometimes the outermost loop is so short that not all threads are utilized:

```
a();
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```



We could try to parallelize the **inner** loop. However, then we will have more overhead in the inner loop, which is more performance-critical, and there is no guarantee that the thread utilization is any better:

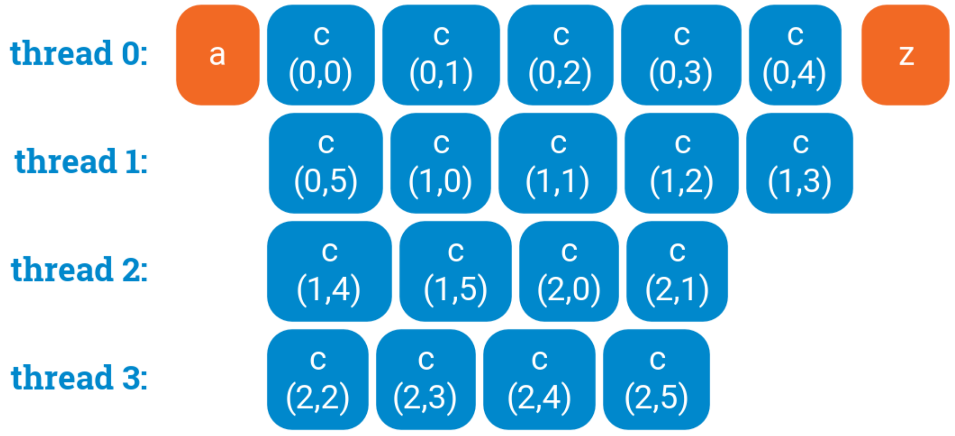
```
a();
for (int i = 0; i < 3; ++i) {
    #pragma omp parallel for
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```



Good ways to do it

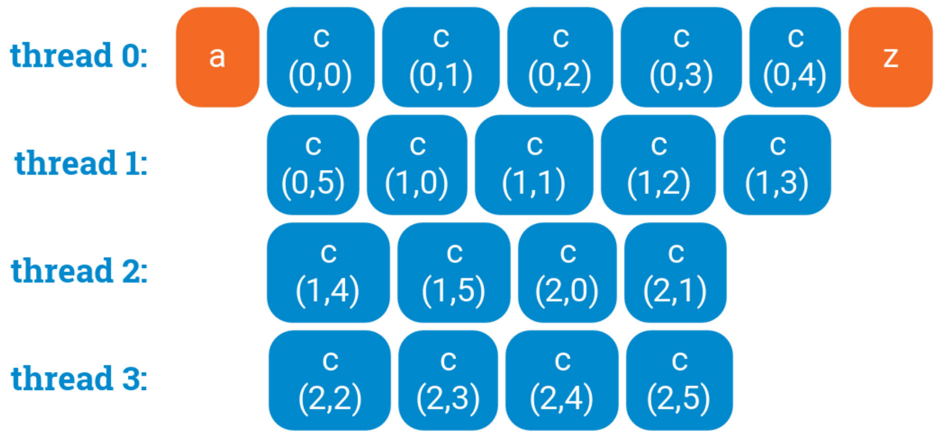
In essence, we have got here 3 × 6 = 18 units of work and we would like to spread it evenly among the threads. The correct solution is to **collapse it into one loop** that does 18 iterations. We can do it manually:

```
a();
#pragma omp parallel for
for (int ij = 0; ij < 3 * 6; ++ij) {
    c(ij / 6, ij % 6);
}
z();
```



Or we can ask OpenMP to do it for us:

```
a();
#pragma omp parallel for collapse(2)
for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```



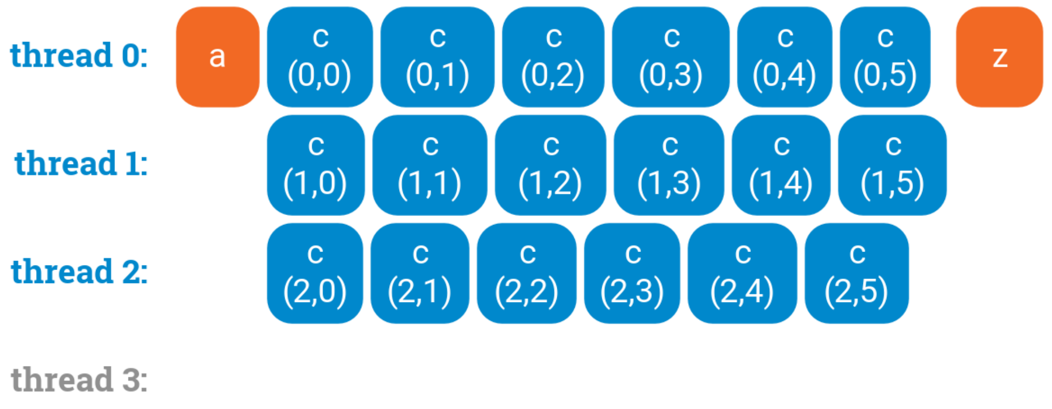
Either of the above solutions are just fine.

Wrong way to do it, part 1

Unfortunately, one often sees failed attempts of parallelizing nested for loops. This is perhaps the most common version:

```
a();
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
    #pragma omp parallel for
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```

This code **does not do anything meaningful**. “Nested parallelism” is disabled in OpenMP by default, and the **second pragma is ignored at runtime**: a thread enters the inner parallel region, a team of only one thread is created, and each inner loop is processed by a team of one thread. The end result will look, in essence, identical to what we would get without the second pragma — but there is just more overhead in the inner loop:



On the other hand, if we tried to enable “nested parallelism”, things would get much worse. The inner parallel region would create more threads, and overall we would have **3 × 4 = 12 threads** competing for the resources of 4 CPU cores — not what we want in a performance-critical application.

Wrong way to do it, part 2

One also occasionally sees attempts of using multiple nested `omp for` directives inside one `parallel` region. This is **seriously broken**; OpenMP specification does not define what this would mean but simply forbids it:

```
a();
#pragma omp parallel for
for (int i = 0; i < 3; ++i) {
    #pragma omp for
    for (int j = 0; j < 6; ++j) {
        c(i, j);
    }
}
z();
```

In the system that we have been using here as an example, the above code thankfully gives a **compilation error**. However, if we manage to trick the compiler to compile this, e.g. by hiding the second `omp for` directives inside another function, it turns out that the program **freezes** when we try to run it.

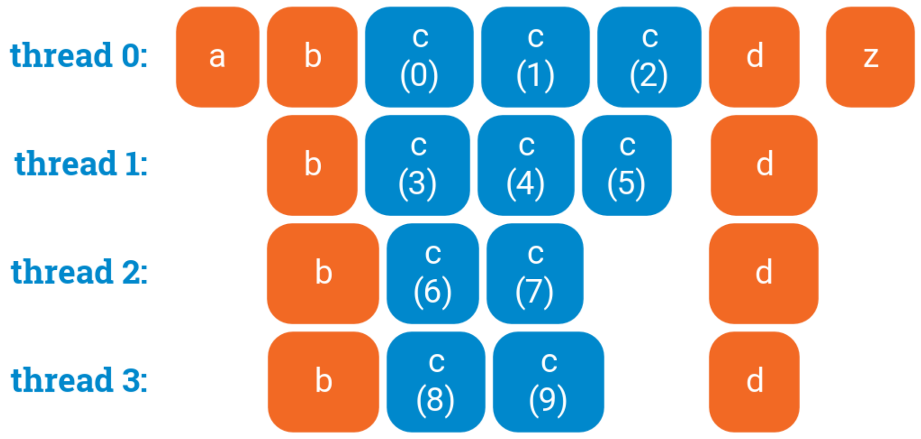
Programming Parallel Computers

Chapter 3: Multithreading with OpenMP

OpenMP parallel for loops: waiting

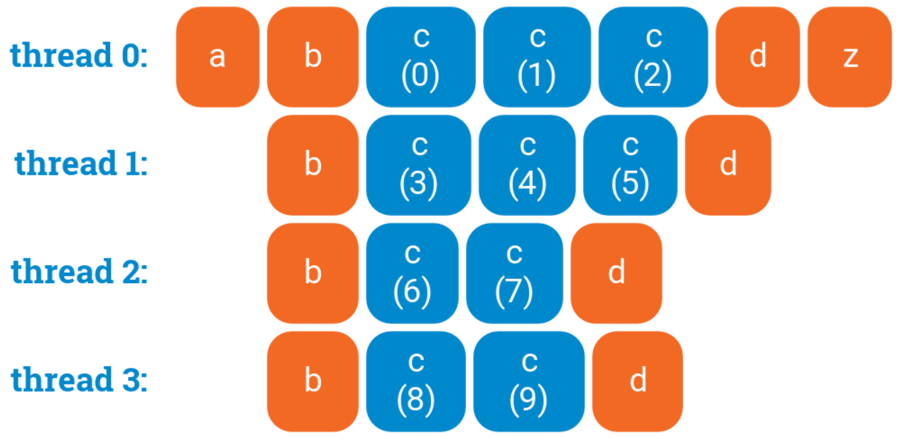
When you use a `parallel` region, OpenMP will automatically wait for all threads to finish before execution continues. There is also a synchronization point **after** each `omp for` loop; here no thread will execute `d()` until all threads are done with the loop:

```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



However, if you do not need synchronization after the loop, you can disable it with `nowait`:

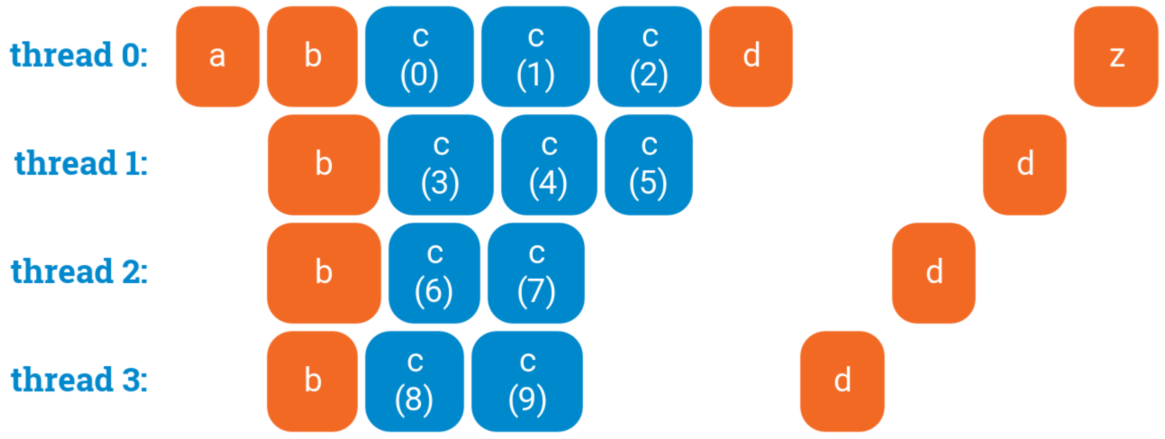
```
a();
#pragma omp parallel
{
    b();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



Interaction with critical sections

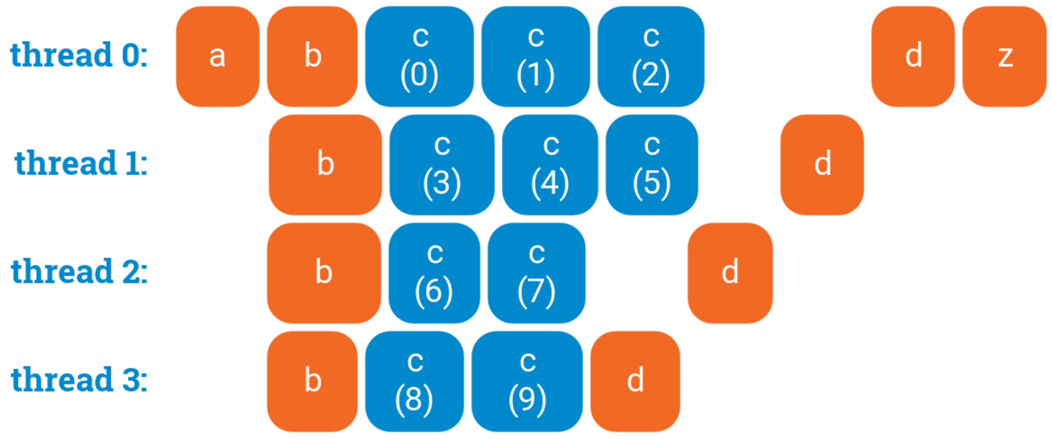
If you need a critical section after a loop, note that normally OpenMP will first wait for all threads to finish their loop iterations before letting any of the threads to enter a critical section:

```
a();
#pragma omp parallel
{
    b();
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    #pragma omp critical
    {
        d();
    }
}
z();
```



You can disable waiting, so that some threads can start doing postprocessing early. This would make sense if, e.g., `d()` updates some global data structure based on what the thread computed in its own part of the parallel for loop:

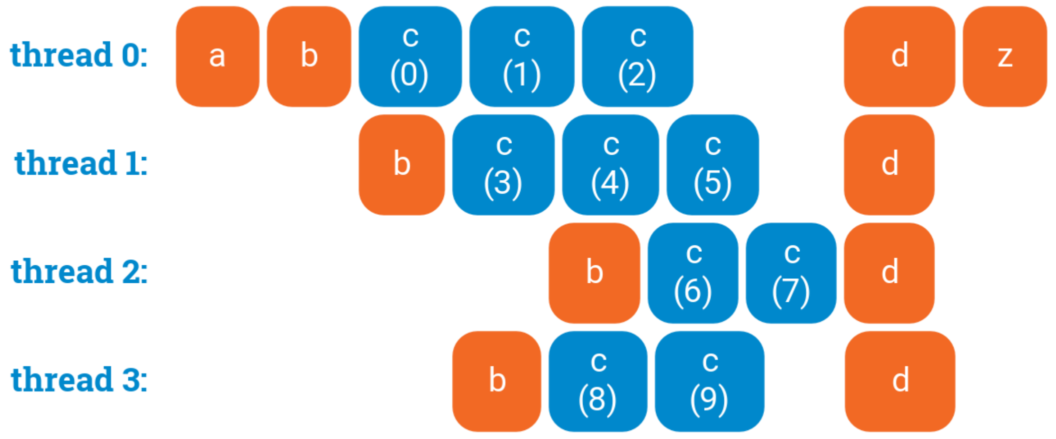
```
a();
#pragma omp parallel
{
    b();
    #pragma omp for nowait
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    #pragma omp critical
    {
        d();
    }
}
z();
```



No waiting before a loop

Note that there is no synchronization point **before** the loop starts. If threads reach the for loop at different times, they can start their own part of the work as soon as they are there, without waiting for the other threads:

```
a();
#pragma omp parallel
{
    #pragma omp critical
    {
        b();
    }
    #pragma omp for
    for (int i = 0; i < 10; ++i) {
        c(i);
    }
    d();
}
z();
```



Programming Parallel Computers

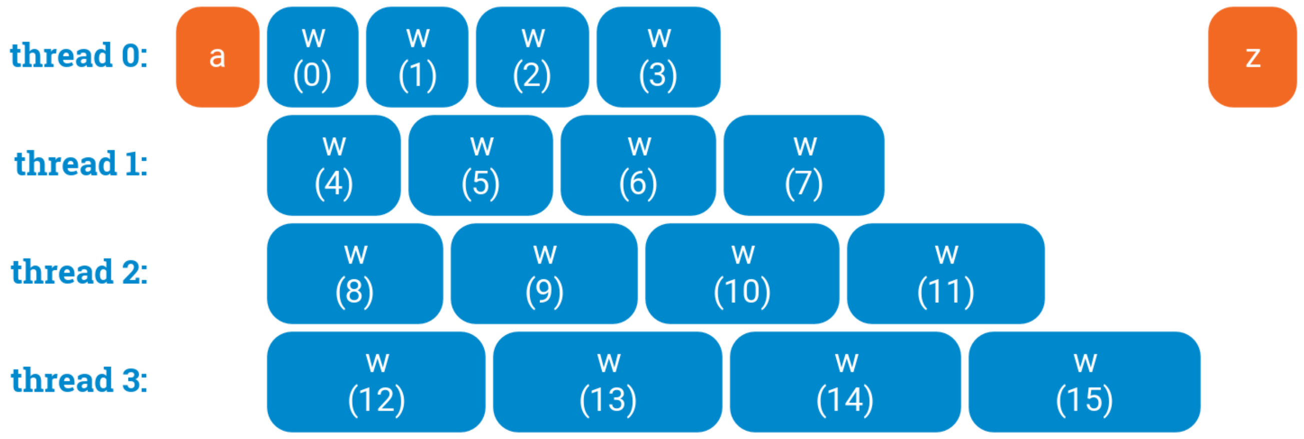
Chapter 3: Multithreading with OpenMP

OpenMP parallel for loops: scheduling

If each iteration is doing roughly the same amount of work, the standard behavior of OpenMP is usually good. For example, with 4 threads and 40 iterations, the first thread will take care of iterations 0–9, the second thread will take care of iterations 10–19, etc. This is nice especially if we are doing some memory lookups in the loop; then each thread would be doing linear reading.

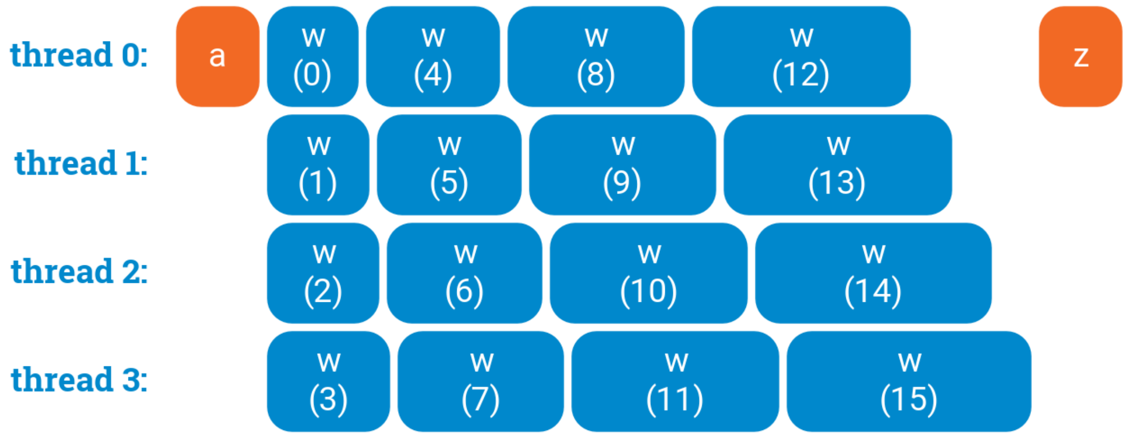
However, things are different if the amount of work that we do varies across iterations. Here we call function `w(i)` that takes time that is proportional to the value of `i`. With a normal parallel for loop, thread 0 will process all small jobs, thread 3 will process all large jobs, and hence we will need to wait a lot of time until the final thread finishes:

```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



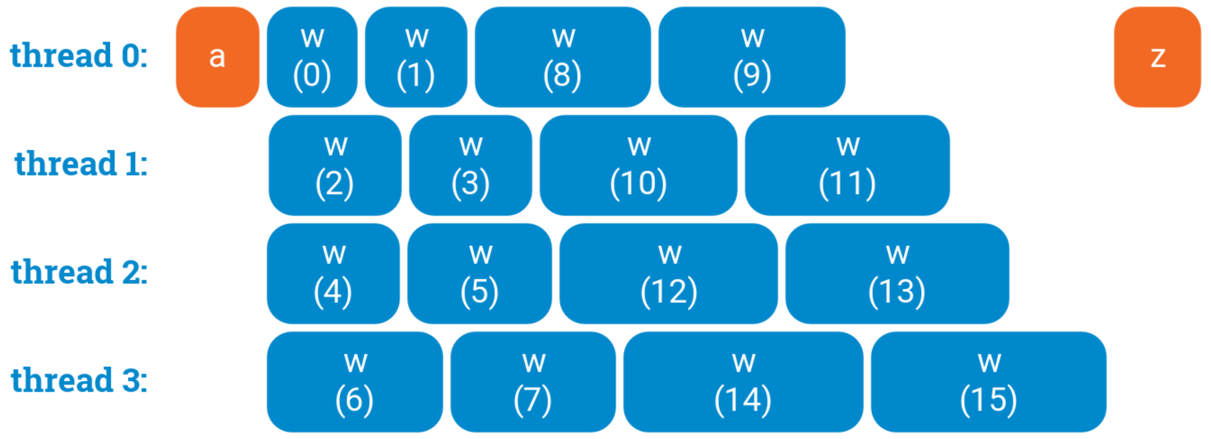
We can, however, do much better if we ask OpenMP to assign iterations to threads in a cyclic order: iteration 0 to thread 0, iteration 1 to thread 1, etc. Adding `schedule(static,1)` will do the trick:

```
a();
#pragma omp parallel for schedule(static,1)
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```



Number “1” indicates that we assign one iteration to each thread before switching to the next thread — we use **chunks** of size 1. If we wanted to process the iterations e.g. in chunks of size 2, we could use `schedule(static,2)`, but in this case it is not useful:

```
a();
#pragma omp parallel for schedule(static,2)
for (int i = 0; i < 16; ++i) {
    w(i);
}
z();
```

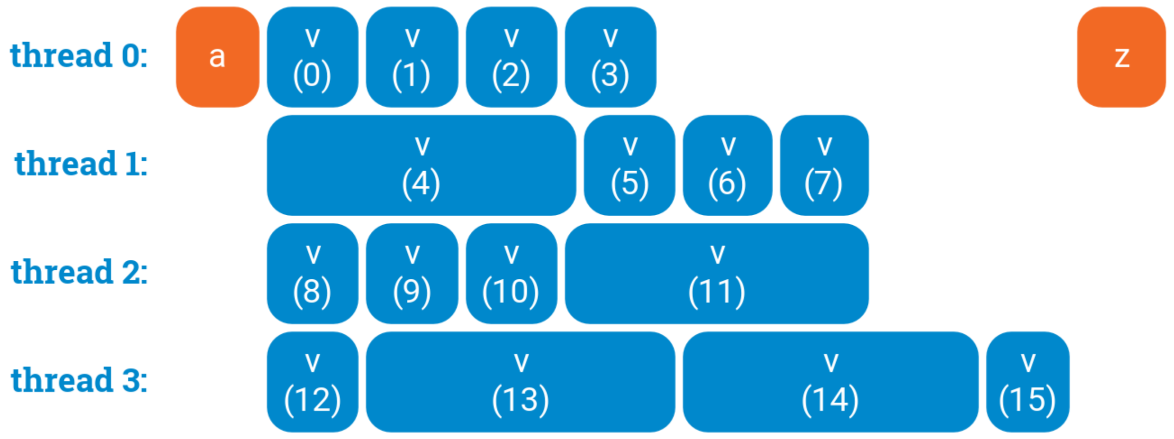


Dynamic loop scheduling

Default scheduling and static scheduling are **very efficient**: there is **no need for any communication between the threads**. When the loop starts, each thread will immediately know which iterations of the loop it will execute. The only synchronization part is at the end of the entire loop, where we just start for all threads to finish.

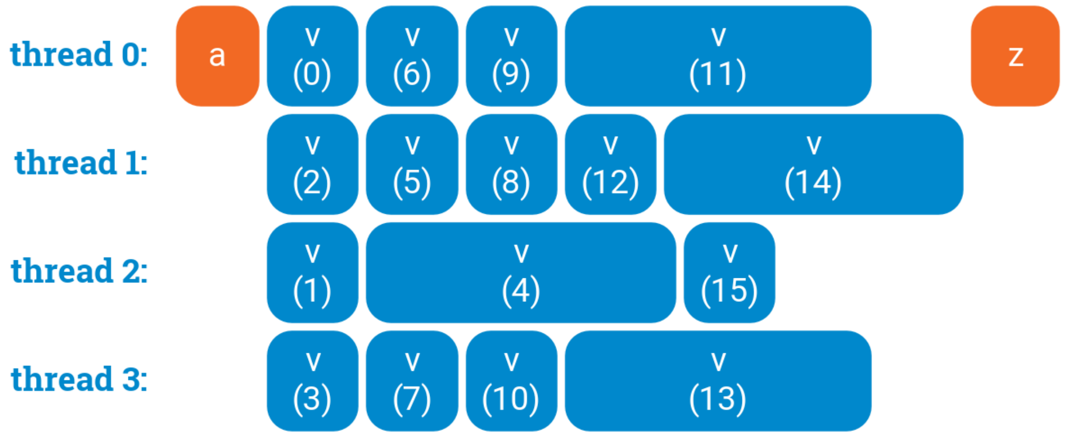
However, sometimes our workload is tricky; maybe there are some iterations that take much longer time, and for any fixed schedule we might be unlucky and some threads will run much longer:

```
a();
#pragma omp parallel for
for (int i = 0; i < 16; ++i) {
    v(i);
}
z();
```



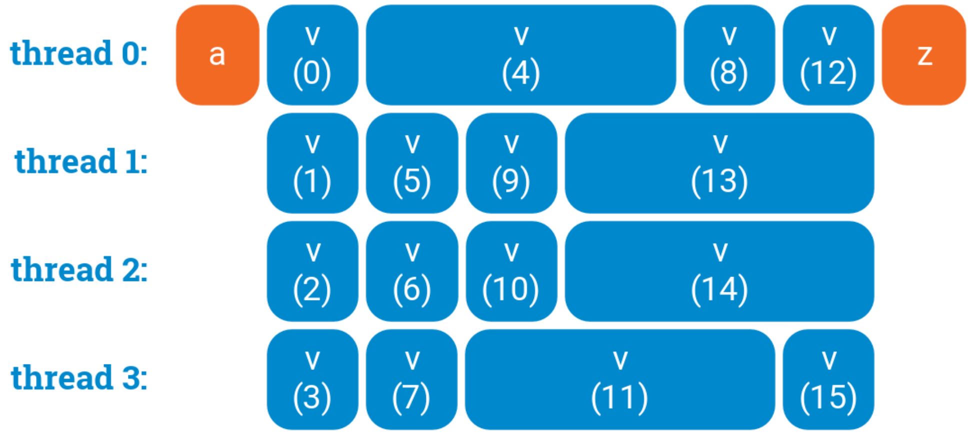
In such a case we can resort to dynamic scheduling. In dynamic scheduling, each thread will take one iteration, process it, and then see what is the next iteration that is currently not being processed by anyone. This way it will never happen that one thread finishes while other threads have still lots of work to do:

```
a();
#pragma omp parallel for schedule(dynamic,1)
for (int i = 0; i < 16; ++i) {
    v(i);
}
z();
```



However, please note that dynamic **scheduling is expensive**: there is some communication between the threads after each iteration of the loop! **Increasing the chunk size** (number "1" in the `schedule` directive) may help here to find a better trade-off between balanced workload and coordination overhead.

It is also good to note that dynamic scheduling **does not necessarily give an optimal schedule**. OpenMP cannot predict the future; it is just assigning loop iterations to threads in a greedy manner. For the above workload, we could do better e.g. as follows:


thread 0:
thread 1:
thread 2:
thread 3:
a
v (0)
v (4)
v (8)
v (12)
z
v (1)
v (5)
v (9)
v (13)
v (2)
v (6)
v (10)
v (14)
v (3)
v (7)
v (11)
v (15)